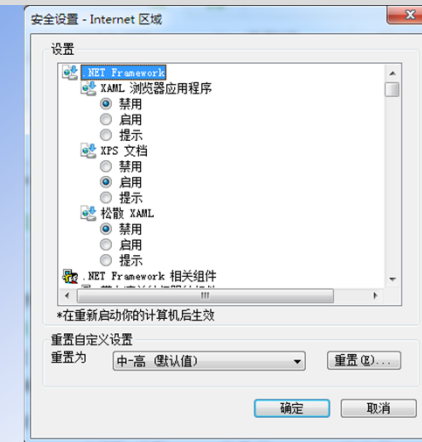


## 两两组合 (Pairwise) 方法

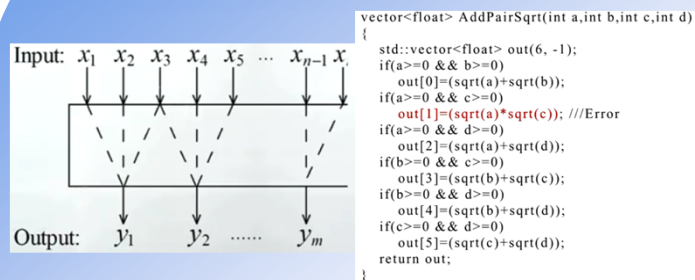
- <http://www.pairwise.org/default.html>
- Combinatorial Test Case Generation
- 大部分缺陷是在两个变量取值冲突的测试时被发现
- 不仅仅是在所有的组合情况下才会发现所有的测试缺陷。这个是“Pairwise Testing”基本原理，不要测试所有的组合，测试所有的“Pairwise”即可
- Pairwise (a.k.a. all-pairs) testing is an effective test case generation technique that is based on the observation that most faults are caused by interactions of at most two factors. Pairwise-generated test suites cover all combinations of two therefore are much smaller than exhaustive ones yet still very effective in finding defects.



IE浏览器安全设定对话框，如果完成所有可能取值的笛卡尔积组合，假设每毫秒完成一个测试，大约需要3300年才可完成所有组合测试。



- 不同变量之间具有协作关系，不同变量之间的组合



## 完全组合?

Input A	Input B	Input C	Input D
A1	B1	C1	D1
A2	B2	C2	D2
			D3

Test all possible combinations of parametric values

### Test cases

Input A	Input B	Input C	Input D
A1	B1	C1	D1
A1	B1	C1	D2
A1	B1	C1	D3
A1	B1	C2	D1
A1	B1	C2	D2
A1	B1	C2	D3
A1	B2	C1	D1
A1	B2	C1	D2
A1	B2	C1	D3
A1	B2	C2	D1
A1	B2	C2	D2
A1	B2	C2	D3

Input A	Input B	Input C	Input D
A2	B1	C1	D1
A2	B1	C1	D2
A2	B1	C1	D3
A2	B1	C2	D1
A2	B1	C2	D2
A2	B1	C2	D3
A2	B2	C1	D1
A2	B2	C1	D2
A2	B2	C1	D3
A2	B2	C2	D1
A2	B2	C2	D2
A2	B2	C2	D3

Number of test cases?

### Pair-wise testing

Test cases

Input A	Input B	Input C	Input D
A1	B1	C1	D1
A2	B2	C2	D3

→

InputA	InputB	InputC	InputD
A1	B1	C1	D1
A1	B1	C2	D2
A1	B2	C1	D3
A2	B1	C2	D3
A2	B2	C1	D2
A2	B2	C2	D1

任意两变量取值组合都出现

- 将被测试应用抽象为一个受到多个因素影响的系统，其中**每个因素的取值是离散且有限的**。两因素（Pairwise）组合测试生成一组测试用例集，可以覆盖任意两个因素的所有取值组合，在理论上可以暴露所有由两个因素共同作用而引发的缺陷。
- 一些路径需要多个因素满足一定取值组合才能被覆盖，然而两因素组合测试不能保证测试用例集可以覆盖这些组合。因此，在**测试资源允许的情况下**，引入多因素组合覆盖有可能进一步提高错误发现率。
- 多因素（T-way, T>2）组合测试可以生成测试用例集，以覆盖任意T个因素的所有取值组合，在理论上可以发现由T个因素共同作用引发的缺陷。
- 《微软的软件测试之道》建议从两因素组合测试开始，逐渐提高组合维度，直至6因素组合测试，因为研究表明6因素组合测试可以发现绝大多数的程序缺陷。但是，随着组合维度的提高，测试用例数呈爆炸式增长。除非测试用例是由测试先知（Test Oracle）自动化执行，否则几乎没有团队能够完成6因素组合测试。在**测试实践中**，**3因素组合测试可能是比较实际的选择**。

### 1-wise/1-way combinatorial testing

Test cases (T=3)

Input A	Input B	Input C	Input D
A1	B1	C1	D1
A1	B2	C2	D1
A2	B1	C2	D1
A2	B2	C1	D1
A1	B1	C2	D2
A1	B2	C1	D2
A2	B1	C1	D2
A2	B2	C2	D2
A1	B1	C1	D3
A1	B2	C2	D3
A2	B1	C2	D3
A2	B2	C1	D3

可  
变  
粒  
度  
组  
合  
测  
试

Interaction Relationship:

$$R = \{ \{ \text{Input A, Input B, input C} \}, \{ \text{input A, input D} \}, \{ \text{input C, input D} \} \}$$

→

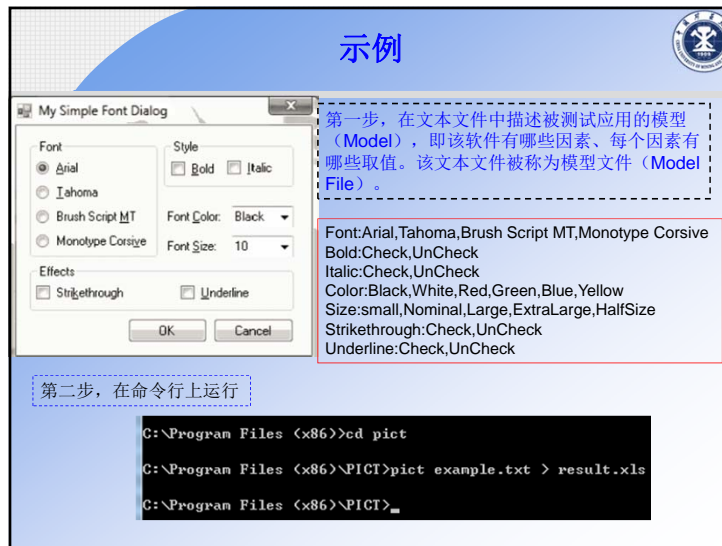
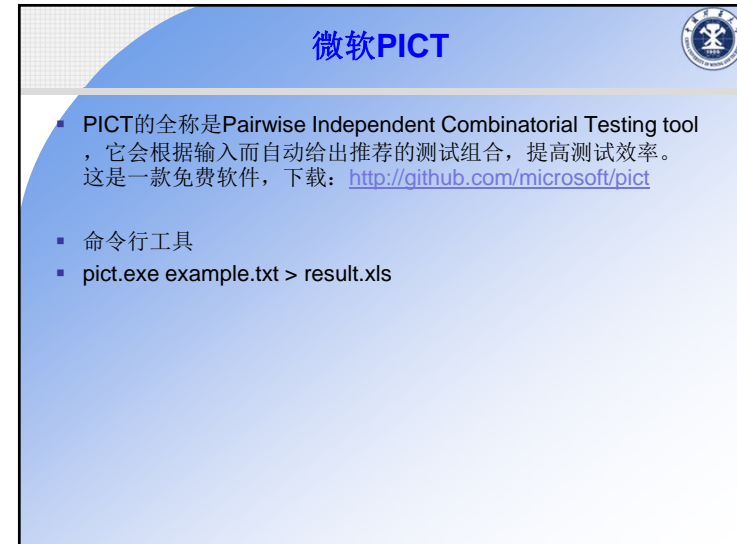
Input A	Input B	Input C	Input D
A1	B1	C1	D1
A1	B1	C2	D2
A1	B2	C1	D3
A1	B2	C2	D3
A2	B1	C1	D2
A2	B1	C2	D1
A2	B2	C1	D2
A2	B2	C2	D1

### 定义因素之间的约束关系

- 在组合测试的基础理论中，各个因素的取值是相互独立的，即因素A的取值不会影响因素B的取值。但是，大多数被测应用的因素之间存在约束关系。以配置测试为例，当因素PLATFORM的取值是x86时，因素RAM的取值就不能是64GB，因为x86 CPU最大只支持4GB RAM。即便试图将组合（PLATFORM: x86, RAM: 64GB）作为负面（negative）测试用例，也是不可行的，因为x86的主板根本插入不了64GB的内存。
- 如果不考虑约束关系，组合测试用例集将包含大量的无效测试用例。这些无效的测试用例，包含一些无效的取值组合，也有可能包含一些有效的取值组合。仅仅删除无效测试用例，会导致最终的测试用例集不能实现两因素或多因素组合覆盖。面对因素之间存在约束关系的被测应用，应该明确定义约束关系，让组合测试工具根据约束来生成有效的测试用例集。
- 在PICT模型文件中加入：
 

```
IF [PLATFORM] = "x86" THEN [RAM] <> "64GB";
IF [OS] = "Win2K3" THEN [IE] >= 6.0;
```

 生成的测试用例集既满足对有效取值组合的覆盖，又不包含无效取值组合。



	A	B	C	D	E	F	G
1	Font	Bold	Italic	Color	Size	Strikethrough	Underline
2	Arial	Check	Check	Green	Nominal	Check	UnCheck
3	Tahoma	UnCheck	UnCheck	White	HalfSize	UnCheck	Check
4	Monotype	UnCheck	Check	Black	ExtraLarge	Check	Check
5	Monotype	Check	UnCheck	Blue	small	UnCheck	UnCheck
6	Brush Script	Check	Check	Green	HalfSize	UnCheck	Check
7	Brush Script	UnCheck	UnCheck	Black	HalfSize	Check	UnCheck
8	Tahoma	UnCheck	Check	Blue	Large	Check	Check
9	Arial	UnCheck	UnCheck	Green	small	Check	Check
10	Arial	Check	UnCheck	Red	Large	UnCheck	UnCheck
11	Brush Script	UnCheck	UnCheck	Blue	Nominal	UnCheck	Check
12	Tahoma	Check	Check	Red	ExtraLarge	UnCheck	UnCheck
13	Monotype	Check	UnCheck	Black	Large	UnCheck	UnCheck
14	Monotype	Check	Check	White	Nominal	Check	UnCheck
15	Brush Script	UnCheck	Check	Red	small	Check	Check
16	Tahoma	Check	UnCheck	Black	small	Check	UnCheck
17	Tahoma	Check	Check	Red	Nominal	Check	UnCheck
18	Arial	Check	Check	Yellow	small	UnCheck	UnCheck
19	Brush Script	UnCheck	UnCheck	Green	Large	Check	Check
20	Tahoma	UnCheck	UnCheck	Yellow	Large	Check	Check
21	Arial	Check	UnCheck	Black	Nominal	UnCheck	Check
22	Monotype	UnCheck	Check	Yellow	Nominal	UnCheck	UnCheck
23	Arial	UnCheck	Check	Blue	HalfSize	UnCheck	UnCheck
24	Arial	Check	Check	White	small	Check	Check
25	Brush Script	UnCheck	UnCheck	White	Large	Check	Check
26	Monotype	Check	Check	Red	HalfSize	Check	UnCheck
27	Brush Script	Check	UnCheck	White	ExtraLarge	UnCheck	UnCheck
28	Brush Script	UnCheck	Check	Yellow	HalfSize	UnCheck	UnCheck
29	Arial	Check	UnCheck	Yellow	ExtraLarge	Check	UnCheck
30	Monotype	Check	UnCheck	Green	ExtraLarge	UnCheck	Check
31	Arial	UnCheck	UnCheck	Blue	ExtraLarge	Check	Check
32	Tahoma	UnCheck	UnCheck	Green	small	UnCheck	UnCheck

原有组合：  
 $4 \times 2 \times 2 \times 6 \times 5 \times 2 \times 2 = 1920$

现： 31

还可以加约束条件

## 小心卫哨(Guard)语句

```
int func(int A, int B, int C)
{
    if (A <= 0) return ERROR; //会“过滤”掉所有A<=0的输入,
    //或者说A是测试中的一个约束
    ...
}
```

- 如果忽视了卫哨语句对执行流的中断, 组合测试用例集将不能达成两因素或多因素覆盖的目标。测试人员要仔细阅读规格说明或源代码, 发现会导致执行流中断的“负面”(Negative)取值。

A: -1, 0, 1  
B: -1, 0, 1  
C: -1, 0,

A	B	C
0	1	-1
1	-1	1
-1	0	-1
1	1	0
-1	-1	0
0	0	1
1	-1	-1
-1	1	1
0	-1	0
1	0	0

- 一种处理方法:

A: -1, 0, 1  
B: -1, 0, 1  
C: -1, 0, 1

A: 1, 10, 100  
B: -1, 0, 1  
C: -1, 0, 1

在生成测试用例集之后, 再加入一条的测试用例 (A: 0, B: 0, C: 0)

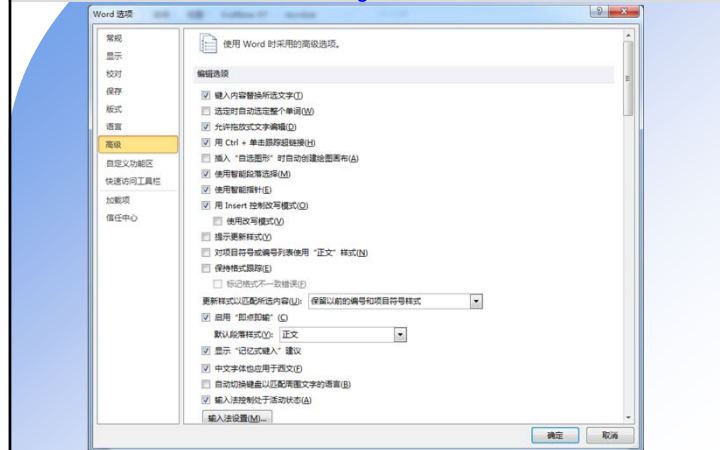
- 另一种处理方法:

A: -1, 0, 1  
B: -1, 0, 1  
C: -1, 0, 1

A: ~0, 1, 10  
B: -1, 0, 1  
C: -1, 0, 1

在PICT的模型中, 用特殊符号 "~" 标记出非法 (invalid) 值。PICT会保证所有有效值的取值组合都会被覆盖, 此外任意非法值与有效值的组合也会被覆盖。

- 如果测试人员不仔细分析被测测试对象, 只依赖组合测试工具, 可能错过有价值的测试用例。
- James Bach 《Pairwise Testing: A Best Practice That Isn't》



```
bool fun(bool a, bool b, bool c, bool d, bool e, bool f, bool g, bool h)
{
    if (a && !b)
        return false;
    return (a && c || b && d) && e && (f && g || !f && h);
}
```

a=true and b=false 是一个约束

- 合并输入变量:

Input A	Input B	Input C	Input D
A1	B1	C1	D1
A2	B2	C2	D2
			D3

Input A	Input B	Input C	Input D
A1	B2		

(Input A Input B)	Input C	Input D
(A1 B1)		D1
(A2 B1)		D2
(A2 B2)	C1	D3

改造输入域模型，分别设计测试用例后合并两组测试用例

Input A	Input B	Input C	Input D
A1	B1	C1	D1
A2	B2	C2	D2
			D3

Constraint

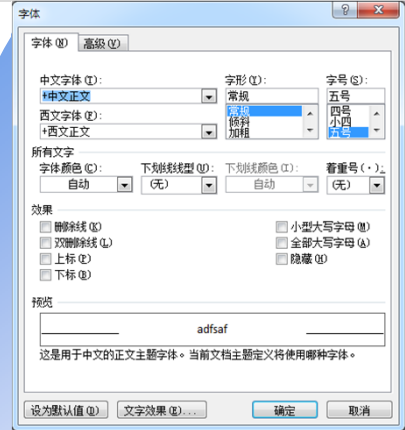
Input A	Input B	Input C	Input D
A1	B2		

改造已有测试用例

(A1,B2,C1,D1) → (\*,B2,C1,D1) (A2,B2,C1,D1)

(A1,B2,C1,D1) → (A1,\*,C1,D1) (A1,B1,C1,D1)

### 含有缺省值的测试



大部分的时候可能不会更改里面的设置，而使用默认的一些值。或即使改变也只是改变其中一个，而不会全部改变。

### 基本选项测试

Equivalence partitioning

Input A	Input B	Input C	Input D
A1	B1	C1	D1
A2	B2	C2	D2
			D3

Default option for each variable

Input A	Input B	Input C	Input D
A1	B2	C2	D3

Test cases

Input A	Input B	Input C	Input D
A1	B2	C2	D3
A2	B2	C2	D3
A1	B1	C2	D3
A1	B2	C1	D3
A1	B2	C2	D1
A1	B2	C2	D2

### 多重基本选项测试

Equivalence partitioning

Input A	Input B	Input C	Input D
A1	B1	C1	D1
A2	B2	C2	D2
			D3

Some variables have more default options

Input A	Input B	Input C	Input D
A1	B2	C2	D1
A1	B2	C2	D3

Test cases

Input A	Input B	Input C	Input D
A1	B2	C2	D1
A1	B1	C2	D1
A1	B2	C1	D1
A1	B2	C2	D2
A1	B2	C2	D3
A1	B2	C2	D3
A2	B2	C2	D3
A1	B1	C2	D3
A1	B2	C1	D3
A1	B2	C2	D1
A1	B2	C2	D2

保证绝大多数情况下是可以正常工作的，对于少数修改配置的也可以保证正常工作。



## 基本概念

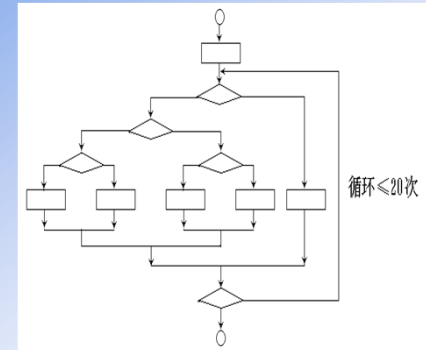
- **逻辑覆盖**：以程序或系统的内部逻辑结构为基础，分为语句覆盖、判定覆盖、判定-条件覆盖、条件组合覆盖等
- **基本路径测试**：在程序或业务控制流程的基础上，分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例。

►逻辑覆盖测试可由弱到强区分为6种覆盖：

1. **语句覆盖**：使程序中的每条可执行语句至少执行一次。
2. **判定覆盖**：使得程序中每个判断的取真分支和取假分支至少经历一次，即判断的真假值均曾被满足。
3. **条件覆盖**：使每个判断中每个条件的可能取值至少满足一次，即每个条件至少有一次为真值，有一次为假值。
4. **判定-条件覆盖**：是将判定覆盖和条件覆盖结合起来，使得判断条件中的每个条件的所有可能取值至少执行一次，并且每个判断本身的可能判定结果也至少执行一次。
5. **条件组合覆盖**：使得所有可能的条件取值组合至少执行一次。
6. **路径覆盖**：覆盖程序中所有可能的路径。

## 全面测试？路径庞大，不可能

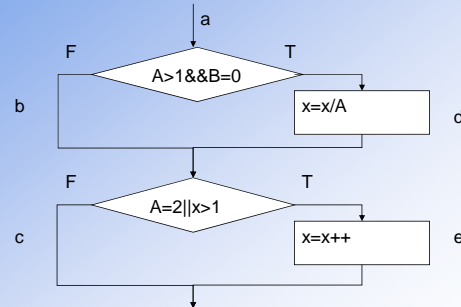
- 对于一个具有多重选择和循环嵌套的程序，不同的路径数目可能是天文数字。给出一个小程序的流程图，它包括了一个执行20次的循环。
- 包含的不同执行路径数达  $5^{20}$  条，对每一条路径进行测试需要1毫秒，假定一年工作  $365 \times 24$  小时，要想把所有路径测试完，需3170年。



## 语句覆盖

- 语句覆盖法的基本思想是设计若干测试用例，运行被测程序，使程序中的每个可执行语句至少被执行一次

A=2,  
B=0,  
x=4,  
路径: ade



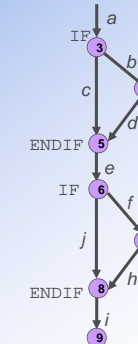
## 语句覆盖可发现的问题

程序源代码

1. dim a, b as integer
2. dim c as double
3. if (a > 0 and b > 0) then
4.     c = c / a
5. end if
6. if (a > 1 or c > 1) then
7.     c = c + 1
8. end if
9. c = b + c

(a, b, c) = (2, 1, 4)

程序控制流程图



## 语句覆盖不能发现的问题

程序源代码

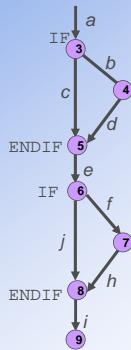
```

1. dim a, b as integer
2. dim c as double
3. if (a > 0 or b > 0) then
4.   c = c / a
5. end if
6. if (a > 1 and c > 1) then
7.   c = c + 1
8. end if
9. c = b + c

```

(a, b, c) = (2, 1, 4)

程序控制流图



- 逻辑运算 (&&, ||) 错误  
判定的第一个运算符 "&&" 错写成 "||", 或第二个运算符 "||" 错写成 "&&", 这时使用上述的测试用例仍然可以达到100%的语句覆盖。
- 循环语句错误
  - 循环次数错误
  - 跳出循环条件错误

- 循环语句例子

```

for(i=0; i<10; i++)
{
  statement;
}
while(x>3)
{
  statement;
}

```

```

for(i=0; i<=10; i++)
{
  statement;
}
while(x>3 && x<7)
{
  statement;
}

```

语句覆盖 = (被执行的语句数量 / 总的语句数量) × 100%  
达到100%的语句覆盖可能很困难, 因为

- 不可达代码;
- 处理错误代码 (处理异常);
- 小概率事件

**【优点】**：可以很直观地从源代码得到测试用例, 无须细分每条判定表达式。

**【缺点】**：由于这种测试方法仅仅针对程序逻辑中显式存在的语句, 但对于隐藏的条件是无法测试的。如在多分支的逻辑运算中无法全面的考虑。语句覆盖是最弱的逻辑覆盖。

语句覆盖率看似很高, 却有严重缺陷:

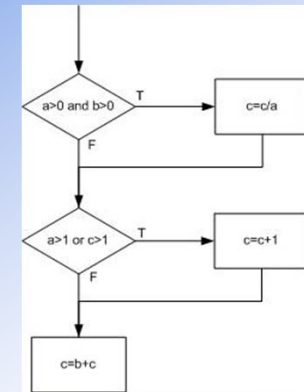
测试用例:	语句覆盖率
if(x!=1) { statements; .....; } 99句	x = 2
else { statement; } 1句	50%的分支没有达到

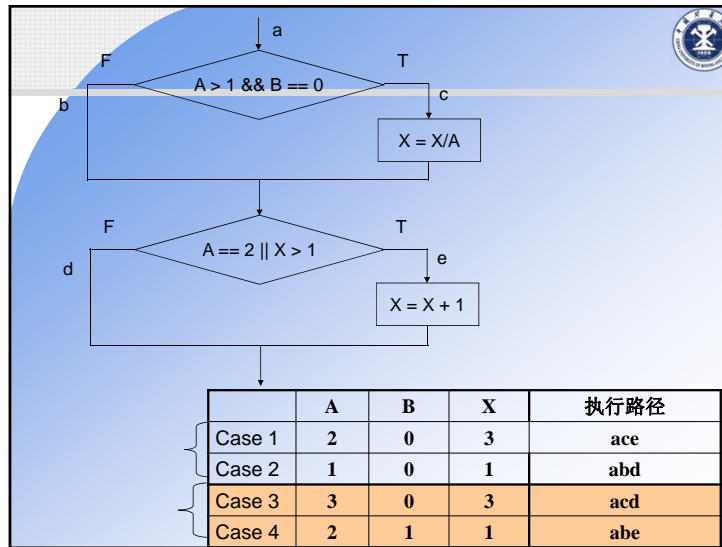
- 测试覆盖率工具: 如TrueCoverage (Numega DevPartner Studio)、PureCoverage (Rational PurifyPlus)

## 判定覆盖

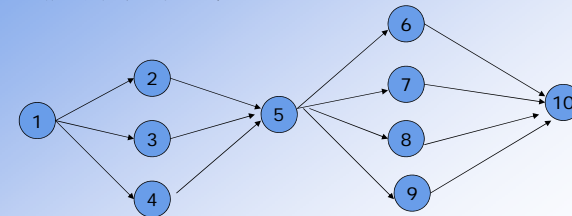
- 判定覆盖法的基本思想是设计若干用例, 运行被测程序, 使得程序中每个判断的true分支和false分支至少经历一次, 即判断真假值均曾被满足。
- 一个判定代表着程序的一个分支, 所以判定覆盖也被称为分支覆盖。
- 覆盖率 = (被执行的分支数量 / 总的分支数量) × 100%

(a, b, c) = (1, 1, 2)  
(a, b, c) = (-1, 1, 0)





- 说明：以上仅考虑了两出口的判断，我们还应把判定覆盖准则扩充到多出口判断（如Case语句）的情况。因此，判定覆盖更为广泛的含义应该是使得每一个判定获得每一种可能的结果至少一次。



- 【优点】：判定覆盖具有比语句覆盖更强的测试能力。同样判定覆盖也具有和语句覆盖一样的简单性，无须细分每个判定就可以得到测试用例。
- 【缺点】：往往大部分的判定语句是由多个逻辑条件组合而成，若仅仅判断其整个最终结果，而忽略每个条件的取值情况，必然会遗漏部分测试路径。判定覆盖仍是弱的逻辑覆盖。

```
if(condition1&& (condition2|| function1()))
    statement1;
else
    statement2;
```

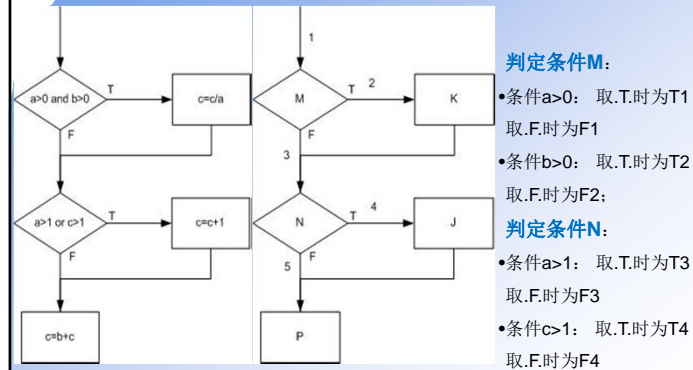
## 条件覆盖

- 条件覆盖的基本思想是设计若干测试用例，执行被测程序以后，要使每个判断中每个条件的可能取值至少满足一次。

$(a > 0 \text{ and } b > 0)$   $\left\{ \begin{array}{l} a > 0 \text{ 原子条件} \\ b > 0 \text{ 原子条件} \end{array} \right.$



### 示例：列出所有条件



### 覆盖所有条件

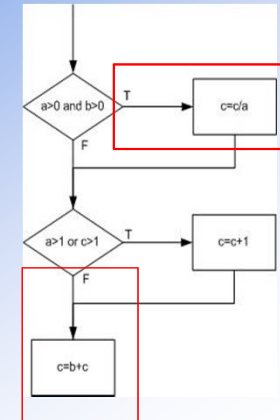
$(a, b, c) = (2, -1, 0)$

T1, F2, T3, F4

$(a, b, c) = (-1, 1, 2)$

F1, T2, F3, T4

但有什么问题吗?



- **【优点】**: 增加了对条件判定情况的测试, 增加了测试路径。
- **【缺点】**: 条件覆盖不一定包含判定覆盖。条件覆盖只能保证每个条件至少有一次为真, 而不考虑所有的判定结果。

### 判定条件覆盖

- 判定-条件覆盖是判定和条件覆盖设计方法的交集, 即设计足够的测试用例, 使得判断条件中的所有条件可能取值至少执行一次, 同时, 所有判断的可能结果至少执行一次。

条件	取值条件	分支	判定条件
$a > 0, b > 0, a > 1, c > 1$	T1, T2, T3, T4	$a > 0 \text{ AND } b > 0$	M=.T. N=.T.
$a \leq 0, b \leq 0, a \leq 1, c \leq 1$	F1, F2, F3, F4	$a > 1 \text{ OR } c > 1$	M=.F. N=.F.

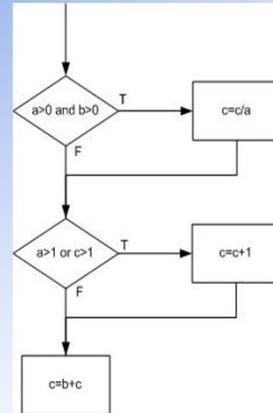
### 示例：覆盖判定/条件

(a, b, c) = (2, 1, 2)

T1, T2, T3, T4

(a, b, c) = (-1, 0, 1)

F1, F2, F3, F4



- 【优点】：能同时满足判定、条件两种覆盖标准。
- 【缺点】：判定/条件覆盖准则的缺点是未考虑条件的组合情况。

分析：从表面上看，判定/条件覆盖测试了各个判定中的所有条件的取值，但实际上，编译器在检查含有多个条件的逻辑表达式时，某些情况下的某些条件将会被其它条件所掩盖。因此，判定/条件覆盖也不一定能够完全检查出逻辑表达式中的错误。

(A>0) && (B>0)，如果A>0为假，则编译器将不再检查B>0这个条件，那么即使这个条件有错也无法被发现。

(A>1) || (C>1)，若条件A>1满足，就认为该判定为真，这时将不会再检查C>1，那么同样也无法发现这个条件中的错误。

### 条件组合测试

- 条件组合覆盖的基本思想是设计足够的测试用例，使得判断中每个条件的所有可能至少出现一次，并且每个判断本身的判定结果也至少出现一次。
- 它与条件覆盖的差别是它不是简单地要求每个条件都出现“真”与“假”两种结果，而是要求让这些结果的所有可能组合都至少出现一次。

### 示例

组合编号	覆盖条件取值	判定条件取值	判定-条件组合
1	T1, T2	M=.T.	a>0, b>0, M取真
2	T1, F2	M=.F.	a>0, b<=0, M取假
3	F1, T2	M=.F.	a<=0, b>0, M取假
4	F1, F2	M=.F.	a<=0, b<=0, M取假
5	T3, T4	N=.T.	a>1, c>1, N取真
6	T3, F4	N=.T.	a>1, c<=1, N取真
7	F3, T4	N=.T.	a<=1, c>1, N取真
8	F3, F4	N=.F.	a<=1, c<=1, N取假

测试用例	覆盖条件	覆盖路径	覆盖组合
输入: a=2, b=1, c=6 输出: a=2, b=1, c=5	T1, T2, T3, T4	P1 (1-2-4)	1, 5
输入: (a,b,c)=(2,-1,-2) 输出: (a,b,c)=(2,-1,-2)	T1, F2, T3, F4	P3 (1-3-4)	2, 6
输入: (a,b,c)=(-1,2,3) 输出: (a,b,c)=(-1,2,6)	F1, T2, F3, T4	P3 (1-3-4)	3, 7
输入: (a,b,c)=(-1,-2,-3) 输出: (a,b,c)=(-1,-2,-5)	F1, F2, F3, F4	P4 (1-3-5)	4, 8

覆盖了所有组合，但覆盖路径有限，1-2-5 没被覆盖

- 【优点】：条件组合覆盖准则满足判定覆盖、条件覆盖和判定/条件覆盖准则。
- 【缺点】：线性地增加了测试用例的数量。

### 修正条件/判断覆盖 (MC/DC)

- MC/DC是DO-178B Level A认证标准中规定的，欧美民用航空器强制要求遵守该标准。
- 修正条件判定覆盖要求在一个程序中每一种输入输出至少得出现一次，在程序中的每一个条件必须产生所有可能的输出结果至少一次，并且每一个判定中的每一个条件必须能够独立影响一个判定的输出，即在其他条件不变的前提下仅改变这个条件的值，而使判定结果改变。
- MC/DC首先要求实现条件覆盖、判定覆盖，在此基础上，对于每一个条件C，要求存在符合以下条件的两次计算：
  - 1) 条件C所在判定内的所有条件，除条件C外，其他条件的取值完全相同；
  - 2) 条件C的取值相反；
  - 3) 判定的计算结果相反。

- 条件true 和 false的独立影响

• To test if (A or B)				
A:	T	F	F	
B:	F	T	F	
• To test if (A and B)				
A:	F	T	T	
B:	T	F	T	
• To test if (A xor B)				
A:	T	T	F	F
B:	T	F	T	F

- MC/DC是条件组合覆盖的子集。条件组合覆盖要求覆盖判定中所有条件取值的所有可能组合，需要大量的测试用例，实用性较差。
- MC/DC具有条件组合覆盖的优势，同时大幅减少用例数。满足MC/DC的用例数下界为条件数+1，上界为条件数的两倍。
- 例如，判定中有三个条件，条件组合覆盖需要8个用例，而MC/DC需要的用例数为4至6个。如果判定中条件很多，用例数的差别将非常大，例如，判定中有10个条件，条件组合覆盖需要1024个用例，而MC/DC只需要11至20个用例。

```
int func(BOOL A, BOOL B, BOOL C)
{
    if(A && (B || C))
        return 1;
    return 0;
}
```

Name	Case 1	case 2	case 3	case 4
A	1	0	1	1
B	1	1	0	0
C	0	0	0	1
ret	1	0	0	1

- 对于条件A，用例1和用例2，A取值相反，B和C相同，判定结果分别为1和0；
- 对于条件B，用例1和用例3，B取值相反，A和C相同，判定结果分别为1和0；
- 对于条件C，用例3和用例4，C取值相反，A和B相同，判定结果分别为0和1。

## 课堂练习

设计测试用例，满足：C/DC、MC/DC

```
程序1
int foo(int x,int y){
    int z=y;
    if((x>5)&&(y>0)){
        z=x;
    }
    return x*z;
}
```

```
程序2
int foo(int x,int y){
    int z=y;
    if(x>5)
        if (y>0){
            z=x;
        }
    return x*z;
}
```

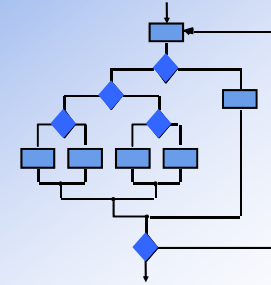
## 基本路径覆盖

- 顾名思义，路径覆盖就是设计所有的测试用例，来覆盖程序中的所有可能的执行路径。
- 完成路径测试的理想情况是做到路径覆盖，但对于复杂性大的程序要做到所有路径覆盖（测试所有可执行路径）是不可能的。
- 在不能做到所有路径覆盖的前提下，如果某一程序的每一个独立路径都被测试过，那么可以认为程序中的每个语句都已经检验过了，即达到了语句覆盖。这种测试方法就是通常所说的基本路径测试方法。
- 独立路径是指程序中至少引入了一个新的处理语句集合或一个新条件的程序通路。**采用流图的术语，即独立路径必须至少包含一条在本次定义路径之前不曾用过的边。
- 测试可以被设计为基本路径集的执行过程，但**基本路径集通常并不唯一**。

测试用例	覆盖路径	覆盖条件	覆盖组合
输入: a=2, b=1, c=6 输出: a=2, b=1, c=5	P1 (1-2-4)	T1, T2, T3, T4	1, 5
输入: a=1, b=1, c=-3 输出: a=1, b=1, c=-2	P2 (1-2-5)	T1, T2, F3, F4	1, 8
输入: a=2, b=-1, c=-2 输出: a=2, b=-1, c=-2	P3 (1-3-4)	T1, F2, T3, F4	2, 6
输入: a=-1, b=2, c=3 输出: a=-1, b=2, c=6	P3 (1-3-4)	F1, T2, F3, T4	3, 7
输入: a=-1, b=-2, c=-3 输出: a=-1, b=-2, c=-5	P4 (1-3-5)	F1, F2, F3, F4	4, 8

## 基本路径覆盖的设计过程

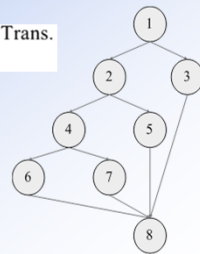
- 基本路径测试方法是在控制流图的基础上，通过分析控制结构的环形复杂度，导出执行路径的基本集，再从该基本集设计测试用例。
- 依据代码绘制流程图
- 确定流程图的圈(环路)复杂度 (cyclomatic complexity)
- 确定线性独立路径的基本集合 (basis set)
- 设计测试用例覆盖每条基本路径



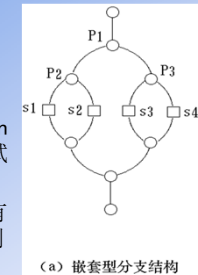
## 环路复杂度

- 环路复杂度是一种为程序逻辑复杂性提供定量测度的软件度量
- 有以下三种方法用于计算环路复杂度:
  - 流图中区域的数量对应于环路的复杂度;
  - 给定流图G的环路复杂度V(G), 定义为 $V(G) = E - N + 2$ , 其中E是流图中边的数量, N是流图中结点的数量;
  - 给定流图G的环路复杂度V(G), 定义为 $V(G) = P + 1$ , 其中P是流图G中判定结点的数量。

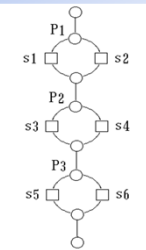
Thomas J. McCabe: A Complexity Measure. IEEE Trans. Software Eng. 2(4): 308-320 (1976)



- 当程序中判定多于一个时, 形成的分支结构可以分为两类: 嵌套型分支结构和连锁型分支结构。
- 对于嵌套型分支结构, 若有n个判定语句, 需要n+1个测试用例;
- 对于连锁型分支结构, 若有n个判定语句, 需要有 $2^n$ 个测试用例, 覆盖它的 $2^n$ 条路径。



(a) 嵌套型分支结构



(b) 连锁型分支结构

基本路径测试并不是测试所有路径的组合, 仅仅保证每条基本路径被执行一次

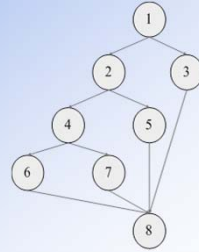


## 示例

```

int IsLeap(int year)
1. {   if (year % 4 == 0)
2.     {
3.         if (year % 100 == 0)
4.         {
5.             if (year % 400 == 0)
6.                 leap = 1;
7.             else
8.                 leap = 0;
9.         }
10.        else
11.            leap = 1;
12.    }
13.    else
14.        leap = 0;
15.    return leap;
16. }

```



- 通过控制流图，计算环路复杂度 $V(G)$ =区域数=4。
- 线性独立的路径集为：
  - 1-3-8
  - 1-2-5-8
  - 1-2-4-7-8
  - 1-2-4-6-8
- 设计测试用例：
 

■ 路径1: 输入数据: year=1999	预期结果: leap=0
■ 路径2: 输入数据: year=1996	预期结果: leap=1
■ 路径3: 输入数据: year=1800	预期结果: leap=0
■ 路径4: 输入数据: year=1600	预期结果: leap=1

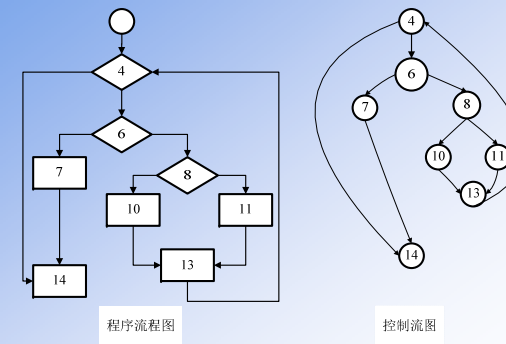
## 示例

```

Void Sort(int iRecordNum, int iType)
1 {
2   int x=0;
3   int y=0;
4   while (iRecordNum-- > 0)
5   {
6     if(0== iType)
7       x=y+2;
8     else
9       if(1== iType)
10        x=y+10;
11      else
12        x=y+20;
13  }
14 }

```

- 第一步，画出程序的控制流图。



- 第二步，计算环复杂度，并确定独立路径。

$$V(G)=E-N+2=4。$$

路径1: 4-14;

路径2: 4-6-7-14;

路径3: 4-6-8-10-13-4-14;

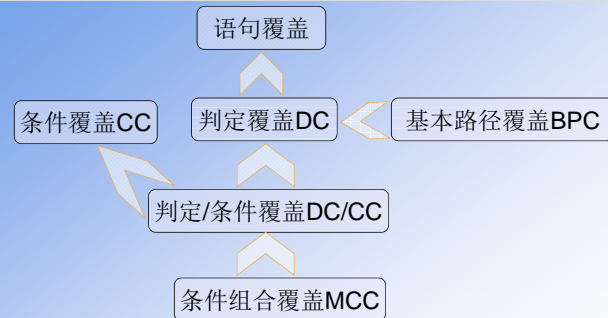
路径4: 4-6-8-11-13-4-14。

- 第三步，导出测试用例

测试用例名称	测试数据	预期结果	测试路径
T1	iRecordNum=0	X=0	路径1
T2	iRecordNum=1 iType=0	X=2	路径2
T3	iRecordNum=1 iType=1	X=10	路径3
T4	iRecordNum=1 iType=2	X=20	路径4

- 第四步，执行测试。

## 小结



- Condition Coverage (CC)
- Decision Coverage (DC)
- Multiple Condition Coverage (MCC)

## 问题?

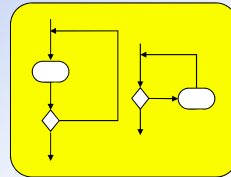
- 代码中循环结构如何测试?
- 从数据输入和结构两方面来考虑

## 带有循环的路径覆盖-1

■ 目标: 在循环内部及边界上执行测试

### 1. 简单循环(迭代次数n)

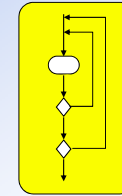
- 完全跳过循环
- 只经过循环一次
- 经过循环两次
- 经过循环 $m$  ( $m < n$ ) 次
- 分别经过循环 $n-1, n, n+1$  次



## 带有循环的路径覆盖-2

### (2) 嵌套循环

- ① 对最内层循环做简单循环的全部测试。所有其它层的循环变量置为最小值;
- ② 逐步外推, 对其外面一层循环进行测试。测试时保持所有外层循环的循环变量取最小值, 所有其它嵌套内层循环的循环变量取“典型”值。
- ③ 反复进行, 直到所有各层循环测试完毕。
- ④ 对全部各层循环同时取最小循环次数, 或者同时取最大循环次数。



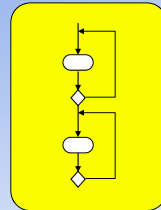
## 带有循环的路径覆盖-3

### (3) 连锁循环

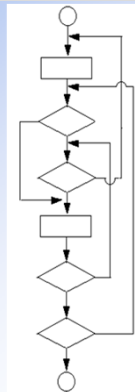
如果各个循环互相独立, 则可以用与简单循环相同的方法进行测试。但如果几个循环不是互相独立的, 则需要使用测试嵌套循环的办法来处理。

### (4) 非结构循环

这一类循环应该使用结构化程序设计方法重新设计。



连锁循环



非结构循环